Write your own Theorem Prover

Phil Scott

27 October 2016

Introduction

We'll work through a *toy* LCF style theorem prover for classical propositional logic. We will:

- review the LCF architecture
- choose a logic
- write the kernel
- derive basic theorems/inference rules
- build basic proof tools
- write a decision procedure

What is LCF?

- A design style for theorem provers.
- Follows the basic design of *Logic of Computable Functions* (Milner, 1972).
- Examples: HOL, HOL Light, Isabelle, Coq.
- Syntax given by a data type whose values are logical terms.
- There is an abstract type whose values are logical theorems.
- Basic inference rules are functions on the abstract theorem type.
- Derived rules are functions which call basic inference rules.

What is Classical Propositional Logic (informally)

- Syntax:
  - Variables $P, Q, \ldots, R$ and connectives $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$
  - Terms/formulas: P, $\neg P$, $P \vee Q$, $P \wedge Q$, $P \rightarrow Q$, $P \leftrightarrow Q$
- Semantics
  - Truth values $\top$ and $\bot$ assigned to variables
  - Connectives evaluate like "truth-functions"; e.g. $\top \vee \bot = \top$
  - Theorems are terms which always evaluate to $\top$ (tautologies)
- Proof Theorems can be found by truth-table checks, DPLL proof-search, or by applying *rules of inference to axioms*.

An inference system for propositional logic

- Given an alphabet α, a term is one of
  - a variable $v \in \alpha$
  - a negation $\neg\phi$ for some formula ϕ (we take $\rightarrow$ to be right-associative)
  - an implication $\psi \rightarrow \phi$ for formulas ϕ and ψ
- A theorem is one of

  Axiom 1 $\phi \rightarrow \psi \rightarrow \phi$ for terms ϕ and ψ

  Axiom 2 $(\phi \rightarrow \psi \rightarrow \chi) \rightarrow (\phi \rightarrow \psi) \rightarrow (\phi \rightarrow \chi)$ for terms ϕ, ψ and χ

  Axiom 3 $(\neg\phi \rightarrow \neg\psi) \rightarrow \psi \rightarrow \phi$ for terms ϕ and ψ

  Modus Ponens a term ψ whenever ϕ and $\phi \rightarrow \psi$ are theorems

The Kernel (syntax)

Formally

Given an alphabet α, a term is one of

- a variable $v \in \alpha$
- an implication $\psi \to \phi$ for formulas ϕ and ψ (we take $\to$ to be right-associative)
- a negation $\neg\phi$ for some formula ϕ

Really Formally

```haskell
infixr 1 :=>:

data Term a = Var a
            | Term a :=>: Term a
            | Not (Term a)
            deriving Eq
```

Theorems

```haskell
axiom1 :: a -> a -> Theorem a
axiom1 p q = Theorem (p :=>: q :=>: p)

axiom2 :: a -> a -> a -> Theorem a
axiom2 p q r =
  Theorem ((p :=>: q :=>: r) :=>: (p :=>: q) :=>: (p :=>: r))

axiom3 :: a -> a -> Theorem a
axiom3 p q = Theorem ((Not p :=>: Not q) :=>: q :=>: p)

mp :: Eq a => Theorem a -> Theorem a -> Theorem a
mp (Theorem (p :=>: q)) (Theorem p') | p == p' = Theorem q
```

Securing the Kernel

```
module Proposition (Theorem, Term(..), termOfTheorem,
                    axiom1, axiom2, axiom3, mp ) where
```

The `Theorem` type does not have any publicly visible constructors. The only way to obtain values of `Theorem` type is to use the axioms and inference rule.

First (meta) theorem

Theorem

For any term P, $P \to P$ is a theorem.

Proof.

Take ϕ and χ to be P and ψ to be $P \to P$ in Axioms 1 and 2 to get:

1. $P \to (P \to P) \to P$
2. $(P \to (P \to P) \to P) \to (P \to P \to P) \to (P \to P)$

   Apply modus ponens to 1 and 2 to get:
3. $(P \to P \to P) \to P \to P$

   Use Axiom 1 with /*phi* and /*psi* to be P to get:
4. $(P \to P \to P)$

Apply modus ponens to 3 and 4. □

First meta theorem formally

Metaproof

```haskell
theorem :: Eq a => Term a -> Theorem a
theorem p =
  let step1 = axiom1 p (p :=>: p)
      step2 = axiom2 p (p :=>: p) p
      step3 = mp step2 step1
      step4 = axiom1 p p
  in mp step3 step4
```

Example

```
> theorem (Var "P")
Theorem (Var "P" :=>: Var "P")

>
```

Issues

- How many axioms are there?

  ```
  axiom1 :: a -> a -> Theorem a
  ```

- How many theorems did we just prove?

  ```
  theorem :: Eq a => Term a -> Theorem a
  ```

- Why could this be a problem for doing formal proofs?

A more(?) efficient axiomatisation

```haskell
(p,q,r) = (Var 'p', Var 'q', Var 'r')
axiom1 :: Theorem Char
axiom1 = Theorem (p :=>: q :=>: p)

axiom2 :: Theorem Char
axiom2 = Theorem ((p :=>: q :=>: r)
                  :=>: (p :=>: q) :=>: (p :=>: r))

axiom3 :: Theorem Char
axiom3 = Theorem ((Not p :=>: Not q) :=>: q :=>: p)

instTerm :: (a -> Term b) -> Term a -> Term b
instTerm f (Var x)     = f x
instTerm f (Not t)     = Not (instTerm f t)
instTerm f (a :=>: c) = instTerm f a :=>: instTerm f c

inst :: (a -> Term b) -> Theorem a -> Theorem b
inst f (Theorem x) = Theorem (instTerm f x)
```

Metaproof again

```
truthThm =
  let inst1 = inst (\v -> if v == 'q' then p :=>: p else p)
      step1 = inst1 axiom1
      step2 = inst1 axiom2
      step3 = mp step2 step1
      step4 = inst (const p) axiom1
  in mp step3 step4
```

```
> theorem
Theorem (Var 'P' :=>: Var 'P')
```

Derived syntax

```haskell
infixl 4 \/
infixl 5 /\

-- | Syntax sugar for disjunction
(\/) :: Term a -> Term a -> Term a
p \/ q = Not p :=>: q

-- | Syntax sugar for conjunction
(/\) :: Term a -> Term a -> Term a
p /\ q  = Not (p :=>: Not q)

-- | Syntax sugar for truth
truth :: Term Char
truth = p :=>: p

-- | Syntax sugar for false
false :: Term Char
false = Not truth
```

A proof tool: the deduction [meta]-theorem

Why did we need five steps to prove $P \to P$. Can't we just use conditional proof?

1. Assume P.
2. Have P.

Hence, $P \to P$.

Deduction Theorem

From $\{P\} \cup \Gamma \vdash Q$, we can derive $\Gamma \vdash P \to Q$.

But Our axiom system says nothing about assumptions!

A DSL for proof trees with assumptions

Syntax

```
data Proof a = Assume (Term a)
             | UseTheorem (Theorem a)
             | MP (Proof a) (Proof a)
          deriving Eq
```

Semantics

```
-- Convert a proof tree to the form Γ ⊢ P
sequent :: (Eq a, Show a) => Proof a -> ([Term a], Term a)
sequent (Assume a)    = ([a], a)
sequent (UseTheorem t) = ([], termOfTheorem t)
sequent (MP pr pr')    =
  let (asms,  p :=>: q)   = sequent pr
      (asms', _) = sequent pr' in
  (nub (asms ++ asms'), q)
```

A DSL for proof trees with assumptions

Semantics

```haskell
-- Send {P} ∪ Γ ⊢ Q to Γ ⊢ P → Q
discharge :: (Ord a, Show a) => Term a -> Proof a -> Proof a

-- Push a proof through the kernel
verify :: Proof a -> Theorem a
```

The implementation of ‘discharge‘ follows the proof of the deduction theorem!

Example with DSL

We want:

```
inst2 :: Term a -> Term a -> Theorem a -> Theorem a

-- ⊢ ¬P → P → ⊥
lemma1 =
  let step1 = Assume (Not p)
      step2 = UseTheorem (inst2 (Not p) (Not (false P)) axiom1)
      step3 = MP step2 step1
      step4 = UseTheorem (inst2 (false P) p axiom3)
      step5 = MP step4 step3
  in verify step5

> lemma1
Theorem (Not (Var 'P') :=>: Var 'P'
           :=>: Not (Var 'P' :=>: Var 'P'))
```

Embedding Sequent Calculus

Assumption carrying proofs

- We'd like to work with proofs of the form $\Gamma \vdash P$ without needing a DSL and a separate verification step.
- We can identify a sequent $P_1, P_2, \ldots, P_n \vdash P$ with the implication $P_1 \to P_1 \to \cdots \to P_n \to P$
- We just need to keep track of n:

```
data Sequent a = Sequent Int (Theorem a)
```

Sequent inference

Modus Ponens on Sequents

Given the sequents

$$\Gamma \vdash P \to Q \text{ and } \Delta \vdash P,$$

we can derive the sequent

$$\Gamma \cup \Delta \vdash Q.$$

Challenge: The union $\Gamma \cup \Delta$ must be computed in the derivation of this rule.

Example

Suppose we want to perform Modus Ponens on

$$P_1, P_2, P_3 \vdash P \to Q \text{ and } P_1, P_3, P_4 \vdash P$$

where $P_i < P_j$ for $i, j \in \{1, 2, 3, 4\}$.

That is, on:

$$(3, P_1 \to P_2 \to P_3 \to (P \to Q))$$

and

$$(3, P_1 \to P_3 \to P_4 \to P).$$

Goal:

$$(4, P_1 \to P_2 \to P_3 \to P_4 \to Q).$$

Computation by conversion

First, use Axiom 1 to add extra conditions on the front of both theorems.

$$P_4 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow (P \rightarrow Q)$$

and

$$P_2 \rightarrow P_1 \rightarrow P_3 \rightarrow P_4 \rightarrow P$$

Using

$$(P \to Q \to R) \leftrightarrow (Q \to P \to R)$$

we have

$$\begin{aligned}
&\boxed{P_4} \to P_1 \to P_2 \to P_3 \to (P \to Q)\\
\leftrightarrow &P_1 \to \boxed{P_4} \to P_2 \to P_3 \to (P \to Q)\\
\leftrightarrow &P_1 \to P_2 \to \boxed{P_4} \to P_3 \to (P \to Q)\\
\leftrightarrow &P_1 \to P_2 \to P_3 \to \boxed{P_4} \to (P \to Q)
\end{aligned}$$

and

$$\begin{aligned}
&\boxed{P_2} \to P_1 \to P_3 \to P_4 \to P\\
\leftrightarrow &P_1 \to \boxed{P_2} \to P_3 \to P_4 \to P
\end{aligned}$$

Computation by conversion

Using

$$(P \to Q \to R) \leftrightarrow (P \wedge Q \to R)$$

we have

$$\begin{aligned}
& P_1 \to P_2 \to P_3 \to P_4 \to (P \to Q) \\
\leftrightarrow & P_1 \wedge P_2 \to P_3 \to P_4 \to (P \to Q) \\
\leftrightarrow & P_1 \wedge P_2 \wedge P_3 \to P_4 \to (P \to Q) \\
\leftrightarrow & P_1 \wedge P_2 \wedge P_3 \wedge P_4 \to (P \to Q)
\end{aligned}$$

and

$$\begin{aligned}
& P_1 \to P_2 \to P_3 \to P_4 \to P \\
\leftrightarrow & P_1 \wedge P_2 \to P_3 \to P_4 \to P \\
\leftrightarrow & P_1 \wedge P_2 \wedge P_3 \to P_4 \to P \\
\leftrightarrow & P_1 \wedge P_2 \wedge P_3 \wedge P_4 \to P
\end{aligned}$$

Computation by conversion

Using axiom 2 and modus ponens, we can then obtain

$$P_1 \land P_2 \land P_3 \land P_4 \to R$$

Then using

$$(P \to Q \to R) \leftrightarrow (P \land Q \to R)$$

we have

$$\begin{aligned}
&P_1 \land P_2 \land P_3 \land P_4 \to R \\
\leftrightarrow &P_1 \land P_2 \land P_3 \to P_4 \to R \\
\leftrightarrow &P_1 \land P_2 \to P_3 \to P_4 \to R \\
\leftrightarrow &P_1 \to P_2 \to P_3 \to P_4 \to R
\end{aligned}$$

Conversions

- A conversion is any function which sends a term ϕ to a list of theorems of the form $\vdash \phi \leftrightarrow \psi$.
- The most basic conversions come from equivalence theorems:
  - Given a theorem of the form $\vdash \phi \leftrightarrow \psi$, we have a conversion which:
    - accepts a term t
    - tries to match t against ϕ to give an instantiation θ
    - returns $\vdash \phi[\theta] \leftrightarrow \psi[\theta]$.
  - For example:
    - the theorem $p \leftrightarrow p$ yields a conversion called `allC`
    - the theorem $(x \leftrightarrow y) \leftrightarrow (y \leftrightarrow x)$ yields a conversion called `symC`
    - the theorem $(P \rightarrow Q \rightarrow R) \leftrightarrow (P \wedge Q \rightarrow R)$ yields a conversion called `uncurryC`

Conversionals

- Functions which map conversions to conversions are called *conversionals*.
- Examples include:

`antC`	converts only the left hand side of an implication
`conclC`	converts only the right hand side of an implication
`negC`	converts only the body of a negation
`orElseC`	tries a conversion and, if it fails, tries another
`thenC`	applies one conversion, and then a second to the results
`sumC`	tries all conversions and accumulates their results

- With these conversionals, we can algebraically construct more and more powerful conversions, implementing our own strategies for converting a term, such as those we need for embedding sequent calculus.

Truth Table Verification informally

- We nominate a fresh proposition variable X and define $\top \equiv X \rightarrow X$.
- Given a proposition, we recurse on the number of other variables.
- Base case: the only variable is X. Evaluate the term according to truth table definitions for each connective. If we evaluate to $\top$, we have a tautology.
- Recursive case: there are n variables other than X. Take the first variable P and consider the two cases $P = \top$ and $P = \bot$. Substitute in these cases and verify that we have a tautology. If so, the original proposition is a tautology.

Truth Table Verification for our Sequent Calculus

- Derive a rule for case-splitting:

$$\frac{\Gamma \cup \{P\} \vdash A \qquad \Delta \cup \{\neg P\} \vdash A}{\Gamma \cup \Delta \vdash A}$$

- Derive theorems for evaluating tautologies:
  - $\top \to \top \leftrightarrow \top$
  - $\top \to \bot \leftrightarrow \bot$
  - $\bot \to \bot \leftrightarrow \top$
  - $\bot \to \bot \leftrightarrow \top$
  - $\neg\top \leftrightarrow \bot$
  - $\neg\bot \leftrightarrow \top$
- Derive $P \vdash P \leftrightarrow \top$ and $\neg P \vdash P \leftrightarrow \bot$

Truth Table Verification for our Sequent Calculus

- Derive a conversion for fully traversing a proposition:

```
depthC :: Conv a -> Conv a
depthC c = tryC (antC (depthC c))
           `thenC` tryC (conclC (depthC c))
           `thenC` tryC (notC (depthC c))
           `thenC` tryC c
```

- Use the conversion and our evaluation rules to fully evaluate a proposition with no variables other than X. If we end up at $\top$, we can then use the derived rule

$$\frac{\Gamma \vdash P = \top}{\Gamma \vdash P}$$

- Wrap up in a verifier (and so claim our axioms complete):

```
tautology :: Term a -> Maybe (Theorem a)
```

Summary

- In LCF, we use a host language (ML, Haskell, Coq etc. . .) to secure and program against a trusted core.
- A bootstrapping phase is usually required to get to the meat.
- We can often follow textbook mathematical logic here, but we do have to worry about computational efficiency.
- We can embed richer logics inside the host logic (e.g. a proof tree DSL or a sequent calculus)
- Combinator languages can be used to craft strategies (for conversion, solving goals with tactics)
- With conversions at hand, problems can be converted to a form where we can implement decision procedures and other automated tools for proving theorems (resolution proof, linear arithmetic, computation of Grobner bases etc. . .)